Julien PILLIS et Harry BROUDEHOUX

Rapport TP3-NF16 : Listes linéaires chaînées

I. Introduction :

L'objectif du TP 3 est de créer un programme qui permet de manipuler des données relatives aux électeurs (nom, numéro de CIN et choix) de l'élection présidentielle. Pour les différentes manipulations des électeurs (différents choix du menu), nous utiliserons l'implémentation des listes d'électeurs sous forme de listes linéaires simplement chaînées en langage C. Pour utiliser cette structure de données, il nous faudra définir les cellules de la liste, représentant les électeurs, ainsi que la liste elle-même, un pointeur vers la première cellule.

II. Liste des structures :

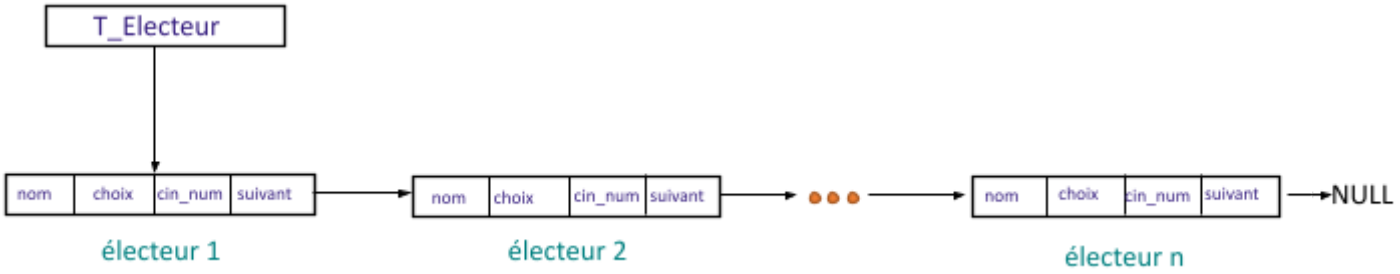
Structure **electeur** :

```
typedef struct electeur
{
    char nom[MAX_LENGTH];
    long cin_num;
    int choix;
    struct electeur *suivant;
}electeur;
```

Structure **T_Electeur** (de type pointeur vers la structure électeur) :

```
typedef struct electeur*T_Electeur;
```

Schéma de la structure de données :



III. Fonctions supplémentaires ajoutées :

Fonction ajout : **void ajout(T_Electeur *tElecteur);**

Cette fonction permet d'ajouter des électeurs à la liste *tElecteur. Elle est appelée dans main.c lors de la sélection de l'option `1. Ajouter des electeurs`. Elle s'avère utile puisqu'elle permet de structurer le code en allégeant le main.c, mais également en simplifiant la démarche d'ajouter des électeurs. En effet, la boucle while de la fonction permet de saisir rapidement les nouveaux électeurs et s'arrête lorsque l'utilisateur le demande.

Fonction réinitialisation : void reinitialisation(T_Electeur *tElecteur);

Cette fonction permet de provoquer la réinitialisation des listes lorsque cela est nécessaire.
Elle n'est utilisée que dans le main.c , lors de la sélection des options 6 et 7. Elle permet simplement de libérer l'espace mémoire allouée d'une liste (*tElecteur) en appelant la fonction `void libereliste(T_Electeur tElecteur)`. Une fois après avoir libéré la mémoire, la fonction paramètre la valeur pointée à NULL (évite les problèmes d'affichage d'une liste vide par exemple). Cette fonction est simple mais permet d'économiser en espace dans le main.c (car beaucoup d'appels) et structure également le code.

Fonction permuter : void permuter(electeur *a,electeur *b);

Cette fonction est utilisée lors de l'appel à la fonction `void triliste(T_Electeur tElecteur)`. Elle permet de permuter deux électeurs dans une liste lorsqu'il est nécessaire (principe du tri à bulles). Ainsi, elle n'est appelée que dans la définition de la fonction triliste. Nous avons choisi d'ajouter cette fonction car elle permet de mieux comprendre l'algorithme du tri à bulle et de mieux le structurer.

Fonction lecture d'un fichier : void lecture_fichier(T_Electeur *tElecteur, char nom_fichier[MAX_LENGTH])

Cette fonction permet de remplir une liste d'électeurs à partir d'un fichier où les lignes sont formatées de la façon suivante : « **Nom : Dumont Num CIN : 14632345 Choix : 3** ». Cela permet de rendre l'ajout beaucoup plus rapide si la liste existe déjà dans un fichier texte. Nous avons choisi d'ajouter cette fonction car elle nous a aidé à effectuer les tests des autres fonctions et elle rend le programme plus performant.

IV. Complexité des fonctions implémentées :

Dans l'étude de complexité des fonctions, nous considérons que ***n*** correspond au nombre d'électeurs de la liste chaînée passé en paramètre.

a. Fonctions principales :

- **T_Electeur creationelecteur(void);**

La fonction ne fait qu'allouer la mémoire au nouvel électeur nouveau et initialiser l'attribut nouveau->suivant à **NULL.** Le nombre d'instructions est constant.
Complexité : $\Theta(1)$

- **void afficheliste(T_Electeur tElecteur);**

La fonction parcourt toute la liste pour afficher les informations relatives à chacun des électeurs.
Ainsi, nous devons parcourir les n électeurs de la liste (boucle while) pour pouvoir les afficher, sauf si la liste pointe vers **NULL** (est vide). (if)
Complexité : $O(n)$

- **void ajoutelecteur(T_Electeur *tElecteur, char nom[MAX_LENGTH], long cin_num, int choix);**

La fonction crée un électeur (Θ(1)) et l'ajoute dans la liste selon l'ordre alphabétique s'il n'est pas déjà présent (trouvelecteur() : Ω (1) et O(n)). Pour cela, nous comparons, avec la fonction `strcmp` (on considère `strcmp` en Θ(1)), le nom du nouvel électeur et le nom de celui pointé par la variable temporaire (un électeur de la liste *tElecteur).
Dans le meilleur des cas, le nom de l'électeur se place avant le nom du premier électeur pointé par la liste *tElecteur (`strcmp(temp->nom, nom) >0`). On le place donc au début de la liste. (if)
Dans le pire des cas, nous devons traverser toute la liste pour ajouter l'électeur (boucle while).
Complexité : Ω (1) et O(n)

- **int comptelecteur(T_Electeur tElecteur);**

La fonction retourne le nombre d'électeurs de la liste. Pour cela, on parcourt l'ensemble de la liste (boucle while) dans tous les cas.
Complexité : Θ(n)

- **int trouvelecteur(T_Electeur tElecteur, long cin_num);**

La fonction retourne la position de l'électeur dans la liste si le numéro de CIN correspondant est trouvé. Sinon elle retourne -1. La fonction affiche également les informations de l'électeurs (O(1) car nombre d'instructions constant).
Dans le meilleur des cas, le premier électeur de la liste est l'électeur recherché. (if)
Dans le pire des cas, l'électeur n'est pas dans la liste ou se trouve en dernière position (on parcourt les n électeurs, boucle while).
Complexité : Ω (1) et O(n)

- **void Supprimelecteur(T_Electeur *tElecteur,long cin_num);**

La fonction supprime l'électeur de la liste, si celui-ci est bien dans celle-ci.
Dans le meilleur des cas, l'électeur à supprimer est le premier électeur de la liste. (trouvelecteur : Ω (1) et O(n))(if)
Dans le pire des cas, l'électeur est introuvable ou est le dernier électeur de la liste (on traverse la liste des n électeurs, boucle while) .
Complexité : Ω (1) et O(n)

- **void decoupeliste(T_Electeur tElecteur,T_Electeur *gauche,T_Electeur *droite,T_Electeur *blanc);**

La fonction ajoute les électeurs (ajoutelecteur : Ω (1) et O(n)) de la liste principale tElecteur dans la liste correspondante. Pour cela, la fonction traverse toute la liste (boucle while).
Dans le meilleur des cas, on ajoute toujours les électeurs au début des listes mais on traverse quand même toute la liste principale.
Dans le pire des cas, on traverse toute la liste principale pour ajouter les électeurs dans une seule liste qui est la plus grande. Ainsi, nous sommes en O(n*max(n1,n2,n3)), avec n1,n2,n3 les tailles de *gauche, *droite et *blanc (dans notre programme ces listes sont vides lors de l'appel à la fonction).
Complexité : Ω (n) et O(n*max(n1,n2,n3))

- **void triliste(T_Electeur tElecteur);**

La fonction effectue un tri à bulles de la liste tElecteur selon le numéro de CIN. On compare chaque élément avec son successeur et on les permute si le numéro de CIN du premier est plus grand que celui de son successeur. Si on effectue au moins une permutation, on re-vérifie si la liste est bien triée..
Dans le meilleur des cas, la liste est déjà triée mais pour vérifier on doit la traverser une fois (while à l'intérieur du do…while).

Dans le pire des cas, il faut permuter n fois tous les éléments de la liste. (permuter : Θ(1)). On vérifie donc n fois que la liste est bien trié (boucle do…while n fois et boucle while n fois)
Complexité : Ω (n) et O(n^2)

- **T_Electeur fusionlistes(T_Electeur tElecteur, T_Electeur tElecteur1);**

La fonction fusionne deux listes et en retourne une nouvelle. Considérons n le nombre d'électeurs de la liste tElecteur et m le nombre d'électeurs de la liste tElecteur1. Pour effectuer cette fusion, nous devons traverser complètement les deux listes (boucles while) pour ajouter (Ω (1) et O(n)) tous les électeurs dans la nouvelle liste. Ainsi, la complexité de la première et de la deuxième boucle sont respectivement O (n^2) et Ω (n), et O (m^2) et Ω (m) . La nouvelle liste formée est alors de taille n+m. On effectue ensuite le tri de cette liste (triliste : Ω (n) et O(n^2)). La complexité de ce tri est alors Ω (n+m) et O($(n+m)^2$) = O(max(n^2,m^2)).
Dans le pire des cas, la liste de la fusion n'est pas triée et on ajoute l'électeur dans ou à la fin de la liste.
Complexité : la règle du maximum implique que l'algorithme est en O(max(n^2,m^2)) et Ω(max(n,m)).

- **int compteGD(T_Electeur tElecteur);**

La fonction traverse toute la liste pour compter le nombre d'électeurs de gauche (boucle while). On vérifie donc le choix de chaque électeur (Θ (1)).
Complexité : Θ (n)

- **void libereliste(T_Electeur tElecteur);**

La fonction libère la mémoire que prend chaque cellule (électeur). Pour cela, il faut traverser l'ensemble de la liste (boucle while). Les instructions à l'intérieur de la boucle sont en O(1).

Complexité : Θ (n)

b. Fonctions supplémentaires :

- **void ajout(T_Electeur *tElecteur);**

Étant donné que c'est à l'utilisateur de signaler la fin de la boucle while, nous ne pouvons pas évaluer la complexité.

- **void reinitialisation(T_Electeur *tElecteur);**

La fonction fait simplement appelle à void libereliste(T_Electeur tElecteur) (Θ(n)) et initialise le pointeur de la liste à **NULL** (O(1)).
Complexité : Θ(n)

- **void permuter(electeur *a,electeur *b);**

La fonction permet d'échanger les données de 2 pointeurs d'électeurs. Son nombre d'instructions est constant, et celles-ci sont toutes en Θ(1).

Complexité : Θ(1)

- **void lecture_fichier(T_Electeur *tElecteur, char nom_fichier[MAX_LENGTH]) ;**

La fonction lit toutes les lignes du fichier et ajoute dans la liste *tElecteur les électeurs de chaque ligne. Si on considère que m est le nombre de lignes du fichier, alors on ajoute (Ω (1) et O(n)) m fois un électeur (tant que la ligne n'est pas vide, on continue, boucle while).
Complexité : Ω (m) et O(m*n)